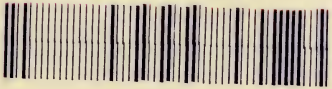


CAMBRIDGE ENGINEERING SERIES
GENERAL EDITOR:
J. F. BAKER, F.R.S.

AN INTRODUCTION TO AUTOMATIC DIGITAL COMPUTERS

AN INTRODUCTION TO AUTOMATIC DIGITAL COMPUTERS

BY

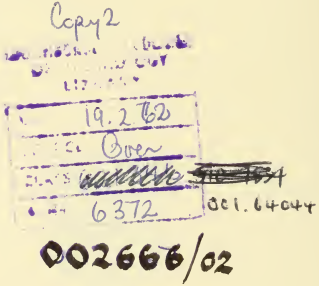
R. K. LIVESLEY

Lecturer in the Department of Engineering,
University of Cambridge

CAMBRIDGE
AT THE UNIVERSITY PRESS
1957

PUBLISHED BY
THE SYNDICS OF THE UNIVERSITY PRESS

Bentley House, 200 Euston Road, London, N.W. 1
American Branch: 32 East 57th Street, New York 22, N.Y.



Printed in Great Britain at the University Press, Cambridge
(*Brooke Crutchley, University Printer*)

CONTENTS

LIST OF PLATES

(*Between pages* 24 *and* 25)

ACKNOWLEDGEMENTS

Plates I(*a*) and II are reproduced by courtesy of Ferranti Ltd., Manchester; Plate I(*b*) by courtesy of the Director of the Mathematical Laboratory, Cambridge University; Plates III and IV(*a*) by courtesy of the English Electric Co. Ltd., Stafford; and Plate IV(*b*) by courtesy of the British Tabulating Machine Co. Ltd.

PREFACE

This brief account of digital computers has been written mainly for engineers and others who are faced with tedious calculations, so that they can judge the possibility of using these machines in their own numerical work. It describes in general terms what digital computers can do and how they are made to do it, and also gives an account of some of the calculations for which they have so far been used. The emphasis is on the application of machines to routine computing work rather than to research.

Since the book is merely an introduction and is not intended for the specialist, I have made no attempt to discuss the design of computing circuits, nor have I tried to make the reader expert in the operation of any particular machine. The reader who wishes to take the subject further will find a more detailed treatment, together with a comprehensive bibliography, in Dr M. V. Wilkes's book *Automatic Digital Computers* (Methuen, 1956), while most computer laboratories issue training manuals for those who wish to become skilled users of their machines. When referring to existing computers I have used the names or initials by which they are generally known. In many cases the full title from which the initials were originally taken has almost been forgotten.

I have avoided as far as possible any detailed discussion of mathematical techniques. It is often assumed that a digital computer can only be used effectively by those with very advanced mathematical training, but this is largely because many of the problems so far solved on computers have been mathematically complex. The preparation of a normal engineering calculation for a digital computer certainly requires care and precise thinking, but it does not usually demand great mathematical skill.

Although I have tried to avoid giving undue prominence to the idiosyncrasies of particular machines, the discussion of engineering problems in Chapter 4 is largely based on my knowledge of work carried out by members of the group using the Manchester University

computer during the period 1951–6. I feel, however, that the conclusions I have drawn are valid for other machines as well.

The book is based on a course of lectures first given at Manchester in the autumn of 1954 to post-graduate apprentices of the Metropolitan-Vickers Electrical Co. Ltd. I am particularly indebted to Professor D. R. Hartree for his comments and suggestions during the preparation of the lectures for publication.

R. K. LIVESLEY

CAMBRIDGE, 1956

Chapter 1

THE ELEMENTS OF PROGRAMMING

1.1. Introduction

The complex calculating machines which have become increasingly well known during the last ten years may be classified as either 'analogue computers' or 'digital computers'.

In an analogue computer, numbers are represented by continuously variable quantities such as lengths, angles or voltages. The electrical or mechanical connexions of the computer are set up in such a way that these quantities satisfy the same mathematical relationships as govern the variables in the problem to be solved. The behaviour of the computer is thus made directly comparable to the behaviour of the physical or mathematical system under consideration.

In a digital computer, on the other hand, numbers are represented by sequences of digits. Although many modern machines do not in fact work in the decimal scale, it is often convenient to think of the numbers which they handle as being composed of decimal digits, stored in registers basically similar to the counting wheels of a desk calculating machine. Since a digital computer can only store rational numbers, the mathematical operations which it can perform are merely those of ordinary arithmetic.

This book is concerned only with digital computers, and for the sake of brevity they will generally be referred to merely as 'computers'. Although these machines are sometimes called 'electronic brains', their electronic nature is in one sense purely incidental. Their importance lies far more in the fact that they are automatic, being able to carry out lengthy sequences of numerical operations without human intervention. Since even the most complex calculations can always be broken down into a series of simple arithmetical steps, digital computers can be applied to many problems unsuitable for analogue solution.

The technique of making a computer carry out a particular calculation is known as 'programming'. This involves first breaking the calculation down into a sequence of arithmetical operations, and then preparing a series of 'instructions' which cause the computer to carry out the required operations in the correct order. Each instruction

specifies a particular step such as multiplication or addition, and it is the programmer's job to see that the instructions are couched in a language or 'code' which the machine can interpret. A whole set of coded instructions, arranged in the correct order, is known as a 'programme'. As there are almost as many instruction codes as there are machines, a programme made up for one computer may not be intelligible to another. It may, however, be used repeatedly on the machine for which it was designed, carrying out any number of similar calculations without the need for further programming.

A simple example of a computer with a permanently built-in programme is a desk calculating machine equipped with automatic division. Once the appropriate button has been pressed a series of subtractions, sign tests and shifts occur without further human intervention. Such an instrument, however, is like a musical box which will only play a single tune. The type of computer described in this book is more like a pianola, which can play any piece of music provided that it is given a set of 'coded instructions' in the form of a suitably punched roll of paper.

It would be out of place in a book of this kind to give a detailed account of the organization and instruction code of a real computer. It is difficult, however, to give an idea of what programming involves without taking particular examples. The remainder of this chapter will be devoted, therefore, to the programming of some numerical examples for an idealized computer with a very simple instruction code.

1.2. A simple calculation and its mechanization

While there are several different ways of solving sets of simultaneous linear algebraic equations, one of the simplest methods is that of successive approximation. Although this technique is only suitable for certain classes of equations, it is very easy to programme for a computer, and is chosen here for that reason.

The equations

$$\left.\begin{aligned} 10x+6y-z&=19,\\ 3x+9y+2z&=27,\\ x-4y+8z&=17,\end{aligned}\right\} \tag{1.1}$$

for instance, may be written in the form

$$x=(19-6y+z)/10, \tag{1.2a}$$

$$y=(27-3x-2z)/9, \tag{1.2b}$$

$$z=(17-x+4y)/8. \tag{1.2c}$$

If an initial 'trial solution' is chosen, such as $x=y=z=1$, equation (1.2a) gives a 'corrected' value for x,

$$x=(19-6+1)/10=1\cdot4,$$

which may be used in equation (1.2b) to give a new value of y,

$$y=(27-3\times1\cdot4-2)/9=2\cdot311.$$

The 'corrected' values of x and y may then be substituted in the third equation (1.2c) to give

$$z=(17-1\cdot4+4\times2\cdot311)/8=3\cdot106,$$

and the whole process repeated with the new values of x, y and z. It is obvious that if the successive values of the variables converge at all, they will converge to the correct solution of the original equations. In fact, successive iterations give

	Initial values	1st iteration	2nd	3rd	4th
x	1	1·400	0·824	0·983	1·002
y	1	2·311	2·035	1·997	1·999
z	1	3·106	3·040	3·002	2·999

The results in the last column are in very close agreement with the exact solution, which is $x=1$, $y=2$, $z=3$.

In the case of a general set of three equations

$$\left.\begin{aligned} a_1x+b_1y+c_1z&=d_1,\\ a_2x+b_2y+c_2z&=d_2,\\ a_3x+b_3y+c_3z&=d_3,\end{aligned}\right\} \tag{1.3}$$

the equations for the nth iteration are

$$\left.\begin{aligned} x_{n+1}&=(d_1-b_1y_n-c_1z_n)/a_1,\\ y_{n+1}&=(d_2-a_2x_{n+1}-c_2z_n)/b_2,\\ z_{n+1}&=(d_3-a_3x_{n+1}-b_3y_{n+1})/c_3.\end{aligned}\right\} \tag{1.4}$$

Alternatively, the process can be thought of as the calculation of a series of successive corrections to the original values of x, y and z. Thus equations (1.4) may be written

$$\left.\begin{aligned} x_{n+1}-x_n&=(d_1-a_1x_n-b_1y_n-c_1z_n)/a_1,\\ y_{n+1}-y_n&=(d_2-a_2x_{n+1}-b_2y_n-c_2z_n)/b_2,\\ z_{n+1}-z_n&=(d_3-a_3x_{n+1}-b_3y_{n+1}-c_3z_n)/c_3,\end{aligned}\right\} \tag{1.5}$$

and it will be noted that the numerators on the right-hand sides of equations (1.5) are merely the differences between the right- and left-hand sides of the original equations (1.3).

In order to carry out this calculation for a given set of coefficients, using equations (1.5), a human being would need a pencil and paper for writing down numbers, some kind of mental or mechanical apparatus for doing arithmetic, and a written or memorized scheme of analysis to follow. This scheme might be

(*a*) Form $(d_1-a_1x_n-b_1y_n-c_1z_n)/a_1$.

(*b*) Add this to x_n to obtain x_{n+1}.

(*c*) Form $(d_2-a_2x_{n+1}-b_2y_n-c_2z_n)/b_2$.

(*d*) Add this to y_n to obtain y_{n+1}.

(*e*) Form $(d_3-a_3x_{n+1}-b_3y_{n+1}-c_3z_n)/c_3$.

(*f*) Add this to z_n to obtain z_{n+1}.

(*g*) Go back to (*a*), testing for convergence on each occasion after the first.

It is apparent that a human being given such a 'programme' of instructions and a set of initial values x_0, y_0, z_0 could carry out the calculation purely mechanically, even if he had never heard of simultaneous equations.

In any pencil-and-paper calculation of this sort, the human computer is involved in three kinds of activity, namely,

(1) Transferring numbers between his paper and his arithmetic machine,

(2) Actually doing arithmetic,

(3) Looking at his list of instructions to see what to do next.

Of these three, it is probable that in many practical cases (2) consumes least time, even when carried out with the simplest type of desk calculating machine. Therefore, even if a high-speed machine which could add or multiply in a millisecond were available its use would not greatly decrease the total time taken by a calculation. A significant saving requires a reduction in the time spent on activities (1) and (3) as well, and this can only be achieved if the whole process is mechanized.

Now consider the features of a machine which could replace the human computer entirely. It would need the following elements:

(i) A 'store', corresponding to the human computer's sheet of paper, in which numbers could be held.

(ii) An 'arithmetic unit' capable of addition, multiplication, etc., together with facilities for transferring numbers between it and the store.

(iii) Means for feeding numbers into the store and displaying results.

(iv) A unit which would control the machine according to a list of 'instructions' supplied by a human operator.

Such a machine would have little advantage over a human being if only a single calculation had to be carried out, as it would probably take as long to prepare the instructions as to do the calculation by hand. However, if the instructions were made independent of the actual numbers appearing in the machine, they could be used repeatedly for carrying out similar calculations with different sets of data, the speed depending only on the rate at which the computer took in and obeyed the instructions.

These ideas form the basis of all automatic digital computers, and were first put forward by Charles Babbage, long before the days of electronics. Babbage envisaged an 'analytical engine' which was to be controlled by holes punched in cards, just as Jacquard had used punched cards to control the weaving of cloth. The specification of this purely mechanical computer contained most of the logical features found in present-day electronic machines, but although parts of it were built it was never completed.

1.3. A simple automatic computer

Although the 'programme' of instructions for solving a set of three simultaneous equations which was developed in the previous section may appear quite simple to a human being, it could only be fed into a machine which understood both algebra and the English language. To make an automatic computer a practical possibility it is necessary to devise a much simpler form for the instructions. To fix ideas, a simple computer using a numerical instruction code will now be described. Although this computer is an imaginary one, it could in fact be built without great difficulty.

The store of the machine consists of a hundred registers which for convenience are numbered 1–100. The number by which a particular register is known is called its 'address', while the decimal number stored in it is referred to as its 'content'. Thus the content of the register whose address is 59 might be the decimal number 3·14159.

The computer is equipped with an arithmetic unit similar to a desk calculating machine, which can carry out addition, subtraction, multiplication and division. This unit contains two special registers known as the multiplier register and the accumulator. The computer is able to read decimal numbers into its store from punched paper tape, and to print numbers on a typewriter. It has two tape-readers,

one for numerical data and one for instructions. The arrangement is shown diagrammatically in fig. 1.

When operating, the machine reads an instruction from the appropriate tape, obeys it, and then proceeds to the next instruction on the tape. The instructions consist of pairs of numbers, each pair consisting of a 'function number' f and an address s. The function

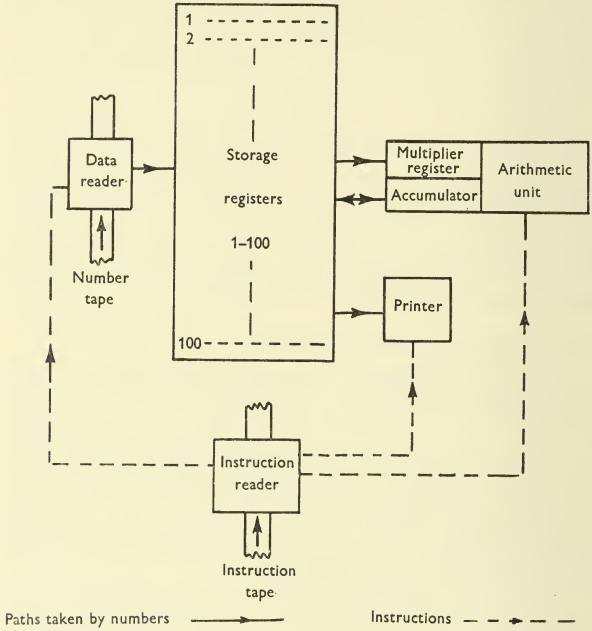


Fig. 1. Arrangement of a simple automatic computer.

number specifies the type of operation to be carried out, while the address specifies the register whose content is to be operated on. The functions described in the following table will be sufficient for a considerable proportion of numerical work. The symbol S indicates the content of the register s referred to by the address section of the instruction, M and A stand for the numbers stored in the multiplier register and the accumulator respectively, while a dash indicates the state of a register after an instruction has been obeyed. The contents

of registers not specifically mentioned remain unchanged, while the number previously stored in a register is obliterated when a new number is transferred to it.

Function number	Effect
1	Read one number into storage register s from the number tape. ($S' =$ tape entry; number tape moves on.)
2	Print the content of register s.
3	Stop. (The address part of this instruction is irrelevant.)
4	Copy the content of register s into the accumulator. ($A' = S$.)
5	Add to the content of the accumulator. ($A' = A + S$.)
6	Subtract from the content of the accumulator. ($A' = A - S$.)
7	Copy the content of register s into the multiplier register. ($M' = S$.)
8	Multiply the content of the multiplier register by the content of the register s, and add the result to the content of the accumulator. ($A' = A + MS$.)
9	As for function 8, but subtract the product from the number in the accumulator. ($A' = A - MS$.)
10	Copy the number in the accumulator into register s and clear the accumulator. ($S' = A$, $A' = 0$.)
11	Clear the accumulator. ($A' = 0$; the address part of this instruction is irrelevant.)
12	Divide the content of register s by the number in the multiplier register and place the result in the accumulator. ($A' = S/M$; the divisor is placed in the multiplier register merely for convenience.)

With this code, a sequence for dividing the number in register 72 by the content of register 49 and planting the result in register 86 would be

Function f	Address s	Effect
7	49	The content of register 49 is copied into the multiplier register.
12	72	The content of register 72 is divided by the number planted in the multiplier register by the previous instruction. The result appears in the accumulator, the previous content of which is obliterated.
10	86	The result is copied into register 86 and the accumulator cleared ready for the next step.

It should be noted that these instructions do not alter the numbers stored in registers 49 and 72.

The instructions for carrying out the process described by equations (1.5) can now be written entirely as a sequence of numbers. If the twelve coefficients $a_1, a_2, a_3; b_1, b_2, b_3; c_1, c_2, c_3; d_1, d_2, d_3$ are punched in that order on the data tape, and are followed by the trial solution x_0, y_0, z_0, the programme for a single iteration might be as follows:

1.1 1.2 ⋮ 1.15	Reading instructions: $a_1 \ldots a_3$ placed in registers 1...3, $b_1 \ldots b_3$ in 4...6, $c_1 \ldots c_3$ in 7...9, $d_1 \ldots d_3$ in 10...12, and x_0 y_0, z_0 in 13, 14, 15.
4.10	$A' = d_1$.
7.13	$M' = x_0$.
9.1	$A' = d_1 - a_1 x_0$.
7.14	$M' = y_0$.
9.4	$A' = d_1 - a_1 x_0 - b_1 y_0$.
7.15	$M' = z_0$.
9.7	$A' = d_1 - a_1 x_0 - b_1 y_0 - c_1 z_0$.
10.16	Result copied into spare register and accumulator cleared.
7.1	$M' = a_1$ (ready for division).
12.16	$A' = (d_1 - a_1 x_0 - b_1 y_0 - c_1 z_0)/a_1 = x_1 - x_0$ from equation (1.5).
5.13	$A' = (x_1 - x_0) + x_0 = x_1$.
10.13	New value of x copied into register 13 and accumulator cleared.
2.13	New value of x printed.
4.11 7.13 9.2 7.14 9.5 ⋮ 10.14 2.14	Similar sequences for calculating y_1.
4.12 7.13 9.3 7.14 9.6 ⋮ 10.15 2.15	Similar sequences for calculating z_1.

The reader will be well advised to work through this sequence in detail. Alternatives are possible, some of which might require fewer instructions. The use of storage registers 1–16 is a purely arbitrary choice.

1.4. The control unit: jump instructions

Machines controlled directly by punched cards or paper tape are quite feasible, and in fact have been produced commercially. They suffer from two drawbacks, however, which have led many computer designers to adopt a different system of control.

In the first place the speed of a machine which reads its instructions direct from punched tape or cards is limited by the mechanical nature of the input device. This will rarely work at more than 50 instructions per second, while electronic circuits can easily be made to operate at more than 1000 instructions per second. In the second place, such a computer cannot easily go back to repeat a series of instructions, nor can it use some numerical criterion to select one of two alternative courses of action unless it has several instruction readers. It is found in practice that facilities for repeating and selecting certain blocks of instructions are extremely useful in designing programmes.

Since instructions can be given the form of numbers, there is no reason why they should not all be placed in the main store of a computer, together with the numerical data, before a calculation is actually commenced. If this is done, it becomes necessary to add to the machine described in § 1.3 a new unit which will be called the 'control' unit, the function of which is to ensure that the instructions are selected from the store and obeyed by the computer in the correct order. This unit contains a register, known as the 'control register', which stores a single number C specifying the address of the instruction being obeyed. It is arranged that when the operation defined by an instruction is completed, the control number C is increased by one, so that the machine passes on to the next instruction. In symbolic terms, each instruction produces the effect $C' = C + 1$.

This method of controlling the computer does not, of itself, increase the overall speed of a calculation, since the instructions presumably have to be fed into the machine from cards or tape in the first place. It is now possible, however, to introduce instructions which alter the content of the control register, thus causing the machine to jump from one point in the programme to another. A simple type of jump instruction is defined as follows:

Function number	Effect
13	Set the content of the control register equal to s. ($C' = s$.)

It will be remembered that after a 'normal' instruction has been carried out, the control number C is increased by one, and the next instruction taken from register $C+1$. The effect of the instruction 13.s is to cause the next instruction to be taken instead from register s. As an example, the following sequence calculates $\frac{1}{1-x}$ from the expansion

$$\frac{1}{1-x}=1+x+x^2+x^3+\ \ldots,$$

where the number 1 is assumed to be available in register 18. The number x (<1) is initially stored in register 19, and this register is subsequently used to accumulate the terms of the series.

Address of instruction	Instruction	Effect
50	7.19	$M'=x$.
51	4.18	$A'=1$.
52	5.19	$A'=1+x$.
53	→ 10.19	On the first cycle, $(1+x)$ is placed in register 19.
54	4.18	$A'=1$.
55	8.19	On the first cycle, $A'=1+x(1+x)$.
56	**13.53**	Return to the instruction in register 53 ($C'=53$).

The reader who follows through this sequence will soon see the power of a few instructions when they form part of a repeated cycle. The second time the instruction in register 53 is reached, its effect is to plant $1+x+x^2$ in register 19. The next cycle calculates $1+x+x^2+x^3$, and so on. A simpler method would be to calculate the individual terms x, x^2, x^3, etc., and sum them, but this would require a few more instructions.

The instruction introduced above can be used to jump forwards as well as back, and its effect can be compared with the jumps which occur in the game of 'Snakes and Ladders'. There is still no choice of next instruction, however, and it is clear that once a loop of instructions has been entered it can never be quitted. A more powerful type of instruction is the 'conditional' jump, whose operation depends on the sign of a certain number, usually the content of the accumulator. A typical example is defined as follows:

Function number	Effect
14	Set the content of the control register to s only if A is positive or zero. ($C'=s$ if $A\geqslant 0$, $C'=C+1$ if $A<0$.)

In other words, the instruction is similar to the previous one, but the jump only occurs if the number currently standing in the accumulator is positive or zero; otherwise the control number merely increases by one and the next instruction is selected in the usual way. A simple sequence for replacing the number x in register 19 by its modulus would be

Address of instruction		Instruction	Effect
50		4.19	$A' = x$.
51		**14.54**	Jump to instruction 54 if $x \geqslant 0$.
52	If	6.19	Subtract $2x$ from A, i.e.
53	$x \geqslant 0$.	6.19	$A' = -x$.
54		10.19	Place $\|x\|$ in register 19.

The conditional jump instruction provides a simple means of repeating a loop of instructions a certain number of times, since a count number can be set before the loop is entered, and reduced by one each time it is traversed. If the number is set equal to $n-1$, after n cycles it will have been reduced to -1, the conditional instruction will cease to cause a jump, and the machine will commence the next part of the calculation.

The reader should now be able to complete the programme for solving a set of three simultaneous equations commenced in §1.3. A suitable test for convergence might be to form the sum of the squares of the corrections to x, y and z, and to test whether or not this exceeded some fixed number ϵ. A jump instruction conditional on the sign of

$$(x_{n+1}-x_n)^2+(y_{n+1}-y_n)^2+(z_{n+1}-z_n)^2-\epsilon$$

could be followed by a 'stop' instruction, so that the machine would halt automatically when a sufficiently accurate solution had been found.

1.5. Modification of instructions

In many calculations it is necessary to carry out the same numerical operations on a series of different numbers. In a computer programme this implies several groups of instructions which, while similar in general pattern, refer to different addresses in the store. In the programme constructed in §1.3, for instance, the instructions for calculating y_1 and z_1 are identical with those for calculating x_1, except that the addresses of the coefficients a_1, b_1, c_1, d_1 are replaced by those

of a_2, b_2, c_2, d_2, etc. In the same way, in the calculation of x_1, the sequence for forming $d_1 - a_1 x_0 - b_1 y_0 - c_1 z_0$ contains a pair of instructions repeated three times, but referring to different addresses each time.

It is clear that if the instructions for calculating x_1 were followed by a sequence which added appropriate numbers to the addresses in certain instructions, they could then be used again to calculate y_1. On the second cycle the modifying sequence would alter the addresses again and the instructions would cause the computer to calculate z_1. In this way considerable savings can be made in the number of instructions required for a given calculation. To sum the contents of fifty consecutive storage locations, for instance, it is only necessary to write down one addition instruction, and arrange for the address in it to be increased by one each time it is used. A conditional jump instruction can be used to count the number of additions in the manner described in §1.4.

In the simple computer which has been described, the task of modifying an instruction and replacing it in its original register requires three instructions, so that modifying sequences will tend to be lengthy if many instructions have to be dealt with. There is also the difficulty of resetting altered instructions if it is necessary to repeat a complete calculation involving several smaller loops of instructions. A system in use on several machines, including the Ferranti computers and the EDSAC, makes such modifications much simpler. In this system, the content of one of a number of subsidiary registers, known as *B*-registers, is added to each instruction before it is obeyed. This is done automatically as the instruction is selected by the control unit, a special part of the instruction specifying which *B*-register is to be chosen. (Instructions which do not require modifying can be arranged to select a *B*-register whose content is zero.) By this means instructions can be made to operate on a series of different addresses merely by altering the content of the appropriate *B*-register. In some computers it is also possible to modify the function number in an instruction, but this is a less useful facility.

B-registers are also useful in counting repetitions, as conditional jump instructions can be introduced which only cause a jump when the content of the appropriate *B*-register is non-negative. Thus a cycle for summing the numbers in the first fifty registers of the simple computer described in this chapter might use the basic instruction 5.1 (add the content of register 1 into the accumulator)

and modify it by a B-register which is initially set to contain 49. The cycle would be:

(1) Clear the accumulator.
(2) Set the B-register to 49.
(3) Obey instruction 5.1 (modified by the B-register).
(4) Subtract one from the B-register.
(5) Test the B-register, and return to (3) if it is non-negative.
(6) Stop.

The first time the control unit selects the instruction 5.1, the actual instruction obeyed is 5.50. The second time it is 5.49, and so on. After it has been obeyed as 5.1, the content of the B-register becomes negative and the cycle comes to an end.

Modified cycles of this type are extremely valuable to the programmer. They are most effective in calculations which are arranged in a systematic form, and for this reason the programmer naturally chooses numerical methods which involve as much repetitive computing as possible. The effect which the properties of a computer have on the choice of suitable numerical methods is discussed further in subsequent chapters.

1.6. Instruction codes in real machines

The simple computer described in this chapter possesses most of the important features found in real machines. The instruction code which has been used is an example of a 'one-address' code, since each single instruction refers to at most one register in the store. Some computers use a two-address code which allows addition operations of the type $A' = S_1 + S_2$ to be carried out by a single instruction, while a three-address code is also possible, allowing operations such as $S_3' = S_1 + S_2$. In general, each instruction in a two- or three-address code does more work, but is more difficult to write down. Some machines, such as the ACE and DEUCE, do not have a control unit of the type described in this chapter, but use an instruction code in which each instruction has a section which specifies the address of its successor.

The design of a digital computer involves a balance between the facilities which the user would like and those which the engineer can reasonably provide. In some cases an instruction operates in what may seem to be an inconvenient way merely to simplify the circuitry. Most of the early computers, for instance, had no division instruction, using instead a sequence of instructions which carried out the process either by repeated subtraction or by a series expansion. The situation

is very similar to that which occurs in desk calculating machines. Facilities for automatic multiplication, division, square-root extraction, etc., are very pleasant for the user, but if they are absent the operations can easily be built up from the simple processes of addition and subtraction.

Most computers have a larger instruction code than the one described in this chapter. It is often necessary to scale a number up or down in order to make it fit into a register, so that an instruction which produces a left or right shift of one or more places is often useful. Many machines also have an 'alarm' instruction, which can be used in place of a 'stop' instruction to summon the operator when a check fails or a calculation comes to an end. The reader will find detailed accounts of the instruction codes used in actual machines in the 'programmer's handbooks' issued by most computing machine laboratories.

Chapter 2

INPUT, STORAGE AND OUTPUT OF NUMBERS

In the previous chapter it was shown how numbers could be used as 'instructions' to control the behaviour of a simple computer. Little was said, however, of the forms taken by instructions and numerical data in the various electronic computers now in use. This chapter discusses the problem of number storage in more detail, and describes some of the methods used in real machines.

2.1. Decimal and binary notation

Most readers will be familiar with the decimal storage registers of a desk calculating machine, or at least with the mileage indicator on a car dashboard. A register of this type consists of a number of counting wheels, each labelled with the digits 0–9 and coupled to its neighbours by a suitable 'carrying' mechanism. There is no theoretical reason why such registers should not be used to store numbers and instructions in an automatic calculating machine, and in fact Babbage's engine was designed to use them, but it is clear that the speed of any computer with mechanical elements is restricted by the inertia of its moving parts. The Harvard Mark I, for instance, which was the first automatic computer actually to be built, had 72 mechanical registers each with 23 decimal counting wheels. It had an addition time of 0·3 sec., and a multiplication time of about 6 sec. Thus, although it was a landmark in the development of automatic computers, it was no faster in carrying out individual operations than an ordinary electric desk calculator. The much higher speeds achieved by later machines are largely due to the development of non-mechanical forms of number storage.

A decimal digit can have ten possible values, so that it requires a storage element which has ten alternative states. Although such elements are used in mechanical calculators, and faster ones using valves called dekatrons can be built for electronic operation, they are inherently slower and more complicated than stores with only two alternative states. A two-state device like a relay, for instance, is much simpler than a multi-state uniselector. Similarly, in transmitting numbers from one part of an electronic machine to another,

it is much easier to convey information by the mere presence or absence of pulses than to use the size of each pulse to distinguish one of ten alternative digits, even though the former system requires more pulses for a given amount of information. The simplest method of feeding numbers into an automatic machine is by using holes punched in paper or card, and here again the presence or absence of a hole defines the two possible conditions of a two-state storage element. Magnetic material, too, can be used most easily as a two-state store, since it is much simpler to measure sense of magnetization than intensity.

For these reasons almost all high-speed computers are based on two-state storage systems. Some, such as the ENIAC, the UNIVAC and the Bell relay machines, nevertheless work on the decimal scale, using groups of two-state elements to represent decimal digits. A group of four such elements, for instance, can take up any one of 16 possible conditions, and ten of these can be associated with the digits 0–9. It might be assumed that a decimal register of this type is wasteful of equipment, since the six remaining storage conditions associated with each group of elements cannot be used directly to signify numbers, but in fact they are often useful for checking purposes. Some machines actually use more than four two-state elements for each decimal digit in order to facilitate checking.

Although it is obviously convenient for the programmer if his computer works in the decimal notation to which he is accustomed, the decimal scale is not the only way of representing numbers. A simpler method of using two-state elements is to work in the scale of two, or the 'binary' scale, as it is often called. A number expressed in binary form consists of a sequence of digits which are either 0 or 1, the convention being that the digit in the nth position is multiplied by 2^{n-1}. This is exactly analogous to the decimal scale, in which the digit in the nth position is treated as being multiplied by 10^{n-1}. The following examples will illustrate the principle:

Decimal form of number	Binary form
$6\ (=6\times10^0)$	$110\ (=1\times2^2+1\times2^1+0\times2^0)$
$18\ (=1\times10^1+8\times10^0)$	$10010\ (=1\times2^4+1\times2^1)$
$273\ (=2\times10^2+7\times10^1+3\times10^0)$	$100010001\ (=1\times2^8+1\times2^4+1\times2^0)$
$3{\cdot}25\ (=3\times10^0+2\times10^{-1}+5\times10^{-2})$	$11{\cdot}01\ (=1\times2^1+1\times2^0+1\times2^{-2})$

Clearly the two possible conditions of a two-state element can be used to represent the digits 0 and 1, and n such elements coupled together will form a register capable of holding any integer between

0 and 2^n-1. Since $2^n \approx 10^{0.3n}$, ten binary digits are approximately equal to three decimal ones. Most computers which work in the binary scale have storage registers of between 30 and 40 binary digits, which corresponds to about 12 decimal places.

The rules of binary arithmetic are very simple, the addition table, for instance, being

$$0+0=0,\ 0+1=1+0=1,$$
$$1+1=10 \text{ (i.e. a 'carry' of 1 to the next place).}$$

It is not necessary, however, for the programmer of a binary machine to be familiar with these rules, since he is rarely concerned with the processes by which the computer carries out his instructions. He need not even know how to translate numbers from decimal into binary form, since the computer can do this for him.

2.2. The sign and size of numbers

Before a row of digits in a binary or decimal storage register can be said to represent a definite number, the sign of the number and the position of the binary or decimal point must be specified or implied.

One method of dealing with signs is to associate with each register an additional two-state element, which records the sign of the number in the register. This system has certain disadvantages, however, and in many computers a different method is used in which negative numbers are represented by their complements. Anyone who has used a desk calculating machine will know that if 1 is subtracted from an empty register the result is a row of 9's. If 2 is subtracted, the result is ...99998, and so on. Thus it is possible to represent -1 by its complement ...99999, etc., provided that a convention is introduced to decide whether a particular sequence of digits is to be interpreted in this way or as an ordinary positive number. Similarly, subtracting 1 from an empty binary register produces a row of 1's, and this may be taken to represent -1. The usual convention is that if a binary register has 0 in its most significant digit position its content is treated as positive, while if the digit is a 1 it represents a negative number in the sense outlined above. It is easy to arrange for negative numbers to be converted into normal sign-and-modulus form on output from the computer.

As in a desk calculating machine, the position of the binary or decimal point in a computer storage register is not usually part of the information stored, but is implied by the arithmetic operations carried

out. Instructions merely operate on signed rows of digits, and it is the programmer's job to see that they have numerical meaning. It may be convenient, for example, to scale numbers so that they are either integers or fractions less than unity, thus placing the binary or decimal point at one end or the other of each register. The programmer must continually guard against the overflow of registers and the loss of significant digits.

The problem of scaling can cause considerable trouble in calculations where it is difficult to assess the magnitudes of numbers. A type of computer which is useful in such cases is one using 'floating point' registers. In 'floating binary' notation, for example, a number x is written in the form $q2^p$, where q is a signed number lying between certain limits and p is a signed integer. A floating binary machine recently built at Manchester University stores numbers in registers of 40 binary digits, each register being divided into two sections. One section of 30 digits stores the number q as a signed fraction whose modulus lies between $\frac{1}{4}$ and $\frac{1}{2}$, while the remaining 10 digits are used for the exponent p. Since p can range from $+511$ to -512, the programmer is virtually free of all scale-factor worries. In the same way, a number can be represented in 'floating decimal' form as $q10^p$. The Harvard Mark II computer, for instance, is a relay machine using floating decimal registers in which the exponent p can vary between $+15$ and -15. Since a floating point machine stores all numbers to the full degree of precision, it may often be more accurate than one using normal registers.

2.3. Binary storage systems

The size of store required for numbers and instructions in a computer depends on the kind of calculation for which it is to be used. Function tabulation, for instance, requires little more storage space than that occupied by the programme, while the solution of 50 linear simultaneous algebraic equations will use 2500 registers for coefficients alone. In general, it seems that a machine which can store about 10,000 numbers is adequate for most reasonable needs, although much useful work has been done by machines with far smaller stores than this.

High-speed storage is expensive, however, and a machine with 10,000 high-speed registers would be very costly both to build and to maintain. Even in the largest problems only a few numbers are manipulated at any one stage, so that most of the registers in such a computer would only be in use for brief periods during a calculation.

Most large computers, therefore, have two types of store—a relatively small high-speed one where the actual computing is done, and a larger auxiliary store which merely acts as a filing system, holding numbers and instructions while they are not in use.

In most automatic computers, the high-speed store has between 50 and 500 registers capable of storing instructions or numerical data. The time taken by the control unit to pick out the content of a particular register is known as the 'access time', and in most modern machines this is of the order of a millisecond.

The auxiliary store of a computer may have many times the capacity of its high-speed store. Numbers contained in the auxiliary store are not directly accessible to the control unit, but blocks of information can be transferred to and from the high-speed store by 'copying' instructions inserted in the programme.

2.31. Types of high-speed store

One of the simplest two-state storage elements is the ordinary electromagnetic relay. These were used in the Harvard Mark II computer, the Bell Telephone Laboratory machines, and in a small computer constructed some years ago at Imperial College. Although reliable, a machine using relays is inherently slow and bulky, and few modern machines use them for storage purposes. They do, however, find application in connexion with auxiliary equipment such as printers, card and tape readers, etc., where speed and compactness are not so necessary.

Another device which can be used to store a binary digit is a valve circuit known as a 'flip-flop'. This has two stable states, and can be triggered from one to the other by applying a pulse of short duration. It is effectively an electronic switch, and can operate at much higher speeds than a mechanical relay. Flip-flops were used for the main store of the ENIAC (1946), the first all-electronic computer. Since each binary digit requires at least two valves, however, a main store built of flip-flops is extremely bulky and expensive; the ENIAC, for instance, contains over 18,000 valves and consumes 150 kW. of power. Later machines use flip-flops extensively in their computing circuits, but other forms of storage are now available which are much more suitable for a large number of digits. The commonest are:

A. *Delay lines*

A form of storage which has been used extensively both in Britain and the United States is the mercury delay line. This consists of

a tube of mercury with a quartz crystal mounted at each end. Electrical pulses are applied to one crystal and generate stress waves in the mercury, which travel down the tube and are picked up at the other end. These pulses are then amplified and fed back to the driving crystal, so that they circulate continuously as long as the power is switched on.

The speed of sound in mercury is about 5000 ft./sec., so that a pulse will take approximately 1 msec. to traverse a delay line 5 ft. long. A delay line of this length can therefore be used to store 1000 pulses, spaced at 1 μsec. intervals, and since each pulse can be made to represent a binary digit this is equivalent to a set of 25 registers of 40 digits each.

The digits stored in a particular register appear as a chain of electrical pulses at the pick-up crystal every time they circulate round the system, and the content of a register may be altered by breaking the connexion between the amplifier and the driving crystal at the appropriate time. Since the contents of registers are only accessible as they pass through the amplifier, the access time to a given register depends on where the digits happen to be in the delay line when they are required. If an instruction affects a number which has just entered a line, it cannot be obeyed until the digits reappear at the other end. In preparing programmes for a delay line machine it is therefore desirable to arrange the timing of numbers and instructions so that the 'waiting time' before each instruction is obeyed shall be as small as possible. This process is known as 'optimum programming'.

Machines which use delay lines for their high-speed store are usually equipped with several lines of different lengths. The DEUCE (pl. III), for instance, has a main store comprising twelve delay lines, each delay line forming 32 registers of 32 binary digits. In addition, there are several shorter lines providing 1, 2 or 4 registers each, which allow rapid access to frequently required numbers and are also used for carrying out arithmetic operations. The basic pulse frequency is 1 Mc./sec., so that the content of a register in one of the longer delay lines appears at 1024 μsec. intervals. Numbers stored in the shorter lines, however, are available at intervals of 32, 64 or 128 μsec.

Mercury delay lines are sensitive to temperature variations and must be thermostatically controlled. In recent years delay lines made of other materials such as nickel have been developed, some of which are less affected by temperature. In a nickel delay line the metal is formed into a thin tube or wire, and stress waves are generated

by passing a current through a coil wound on one end. The magnetic field causes a distortion of the material known as magneto-striction, and this causes a wave to be propagated down the delay line. A pick-up coil at the other end operates in a similar manner. Nickel delay lines have been used in several small computers, as they are light and can easily be made into plug-in units.

B. *Cathode-ray tubes*

A cathode-ray tube forms a convenient means of displaying the contents of a binary register, and it can also be used to store binary digits. The 0's and 1's can be represented either by dots and dashes, or by faint and bright dots on the tube's fluorescent screen (pl. I*a*). The electrons which cause the emission of light also produce a distribution of surface charge on the screen, and this distribution can be maintained by using the phenomenon of secondary emission associated with fluorescent material. When acting as storage device, the cathode-ray tube has a pick-up plate mounted in front of its screen, which is scanned repeatedly by an electron beam in a manner similar to that which occurs in a television tube. As the beam traverses the screen, the digits stored in the form of surface charges produce a series of pulses on the pick-up plate, and these pulses can be used to regenerate the charge pattern which would otherwise gradually leak away.

A single cathode-ray tube can be used to store a large number of digits. In the Ferranti Mark I computer, for example, the high-speed store consists of eight cathode-ray tubes, each storing 1280 binary digits arranged as 32 40-digit registers. The cathode-ray tube has an advantage over the delay line in that all registers are immediately accessible. To scan a particular register it is only necessary to switch the electron beam to the appropriate point on the tube's screen.

C. *Magnetic cores*

Both the delay line and the cathode-ray tube are regenerative systems, requiring a power supply merely to keep the stored information circulating. A type of binary element which can store a single digit without the need for external power is the magnetic core. This consists of a small toroid of moulded ferrite, a material which has an almost rectangular hysteresis loop. The material may be put into one of the two possible states of remanent magnetism by applying a current pulse of the appropriate sense to a coil linking the core. These two states may be used to represent the digits 0 and 1, and the number stored may be 'read' by examining the voltage pulse

produced in a second winding when a current pulse of standard direction is applied to the first.

The cores used in practice are normally about 2 mm. in diameter, so that a large number of digits can be stored in a very small space. The 'writing' and 'reading' coils are often merely single wires which thread the cores and also support them in space (pl. I*b*), and the change-over time from state '0' to state '1' is of the order of a microsecond. Magnetic cores were first used in a store of 1024 16-digit registers built for the 'Whirlwind' computer at M.I.T. in 1953, and more recently they have been used in EDSAC II and the Ferranti Mark II computer. It seems likely that they will very soon replace all other types of store, at least for really high-speed machines.

2.32. Auxiliary storage

Magnetic material provides by far the commonest type of auxiliary storage. It may be used either in the form of a 'magnetic drum', or as tape or wire.

A 'magnetic drum' (pl. IV*a*) consists of a metal cylinder with a magnetizable surface, rotating in synchronization with the rest of the computer. The surface of the drum is divided into 'tracks', spaced along the axis of rotation, each track being equipped with a 'writing' and a 'reading' head. The operation is very similar to that of a tape recorder, the contents of a complete track being transferred to or from the high-speed store in one rotation of the drum. As a typical example, the Ferranti Mark I computer has a drum 10 in. in diameter and 12 in. long rotating at 1955 r.p.m. There are 256 tracks along the length of the drum, each track storing 2560 binary digits, corresponding to the contents of 64 40-digit registers. The contents of a track can be read from the drum to the high-speed store in 36 msec., while the corresponding time for writing on the drum is 63 msec. No facility exists for the transfer of the contents of single registers, but this is not a noticeable drawback in practice.

Transfer instructions appear in programmes along with the ordinary arithmetic instructions. A transfer instruction specifies a track number, a section of the high-speed store, and whether the transfer is to 'read from' or 'write onto' the drum. In some computers, such as the DEUCE, the calculation can be continued without interruption, provided that it does not use the relatively small part of the store concerned in the transfer during the transfer period.

While magnetic drums have been extensively used for auxiliary storage in large computers, they often form the entire store of smaller

machines. Drum computers are relatively cheap to build, and while they may not be as fast as machines with other types of storage, they are usually very reliable. When used as a main store, the magnetic drum suffers from the same disadvantage as the delay line, since the content of a particular register is only accessible once every revolution, but this can be mitigated to some extent by optimum programming techniques. A typical drum computer is the HEC 4E, which has a drum store of 1024 registers, each holding 40 binary digits. The drum revolves at 3000 r.p.m., so that the access time to a particular register is 20 msec. or less.

Magnetic tape has been used as an auxiliary store, notably on the UNIVAC and on the EDSAC. Its main disadvantage would seem to be that not all blocks of information are equally accessible—it might on certain occasions be necessary to run through a whole reel of tape to reach a particular block of numbers. It is possible, however, to attach several tape units to one computer, and the total storage available can be made extremely large.

The input-output medium of a computer can also be used as a large-capacity auxiliary store. If a complete problem is too big for a machine, for instance, intermediate results can be punched out on cards or on paper tape, and fed back through the input device at the appropriate time.

2.4. Input-output equipment

For the sake of reliability and economy, many computers use standard or modified commercial data-processing equipment for the input and output of information. The two commonest input media are punched cards and teleprinter tape (pl. IV*b*) but magnetic tape and wire have also been used. Small items of information can also be fed into a computer by manually operated switches.

2.41. Punched cards

Hollerith cards store information in the form of 12 rows of 80 binary digits each, a hole in a digit position signifying a '1', while a space signifies a '0'. When used in conjunction with an automatic computer, each row of the card is usually made to correspond to a single binary register; this means that part of the card may be wasted (depending on the number of digits in the computer's registers), but this is of little consequence.

A fast card reader can operate at a speed of about three cards per second, corresponding to over 2000 binary numbers per minute, while

a standard output punch works at a slightly lower speed. Results may thus be obtained rapidly in card form and printed out elsewhere without interfering with the computer. A normal computer installation will include standard Hollerith equipment for preparing, sorting, copying and tabulating cards.

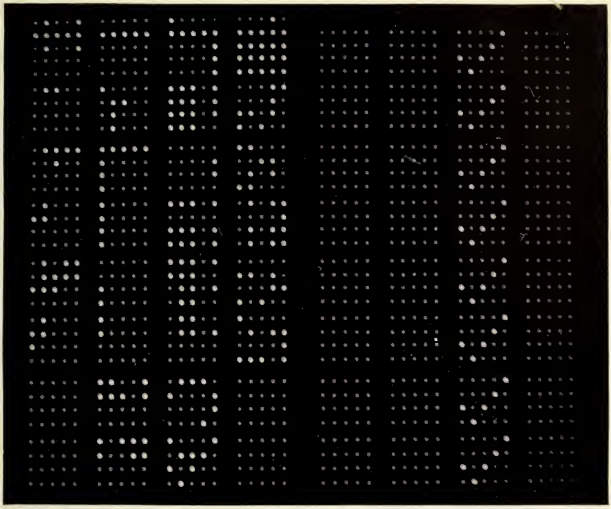
It would be a very tedious manual job to convert all data into binary form and all results back into decimals. It is possible, however, to punch numbers in a decimal code, using the various rows of a card to represent different decimal digits. This pattern of holes is read by the computer in the usual way, and converted into a binary number by a short sequence of instructions supplied by the operator. In the same way a binary-decimal output sequence can be made to produce results in decimal form. In practice, all the necessary calculations can be done in the intervals between the reading or punching operations, so that no machine time is lost. Finally, the decimal numbers on the cards can be printed on a standard Hollerith tabulator. This facility is only valuable if the results are required for human inspection. If they are merely to be fed back into the computer the numbers are best left in binary form.

2.42. Teleprinter tape

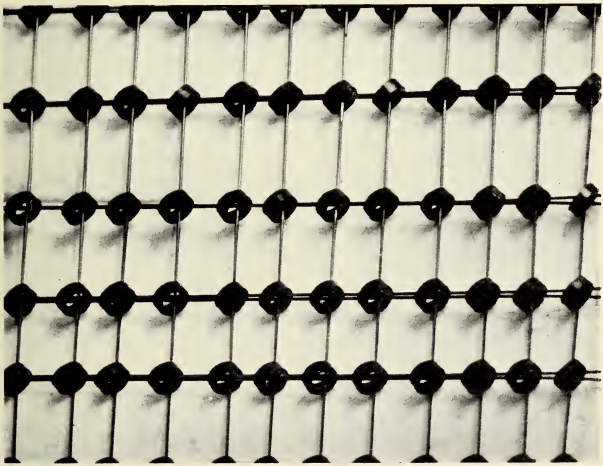
Another common input medium is standard teleprinter tape. This is paper tape with five or six positions in which holes can be punched across its width. Each row of a five-hole tape can store a 5-digit binary number, and when used for telegraphy each of the 32 possible digit combinations is made to represent a letter of the alphabet or other symbol. This means that if standard equipment is used to produce computer input tapes, instructions and data must be written in a form corresponding to the characters on a teleprinter keyboard.

A feature of standard teleprinter tape which is sometimes a disadvantage is the fact that a single row of holes is not long enough to specify an instruction. In the Ferranti Mark I computer, for instance, the input medium is 5-hole paper tape, while the instructions are 20-digit numbers. This means that each instruction appears as four rows of holes on the input tape, and a special 'input programme' is required to assemble the instructions and plant them in the computer's store. Teleprinter tape, however, is quite suitable for presenting decimal numbers to a computer, since each decimal digit can be conveniently represented by a single row of holes.

Since paper tape can store far fewer digits per row than punched cards, satisfactory input speeds can only be achieved by running the

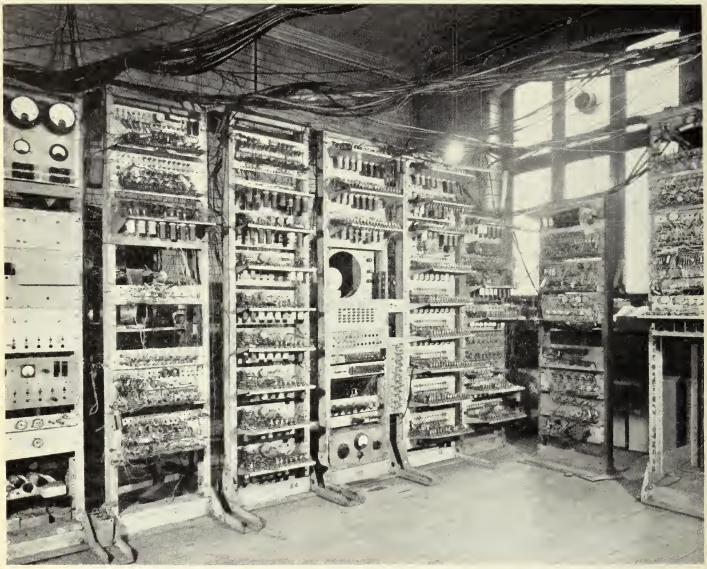


(*a*) Binary digits displayed on a cathode ray tube



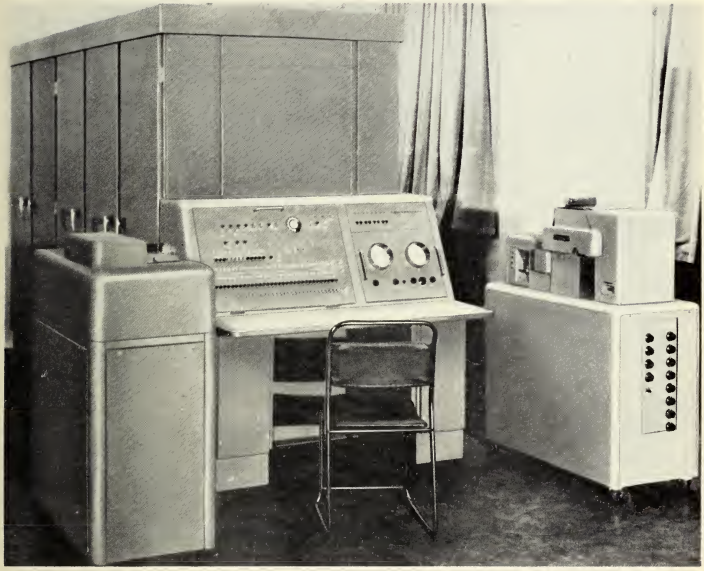
(*b*) Part of a magnetic core storage system

PLATE II

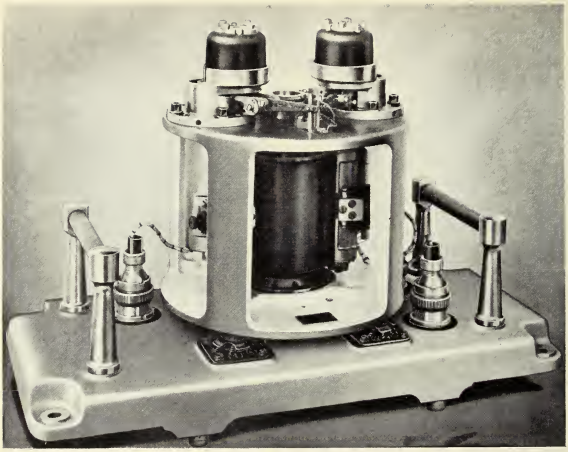


The first Manchester computer, built in 1948

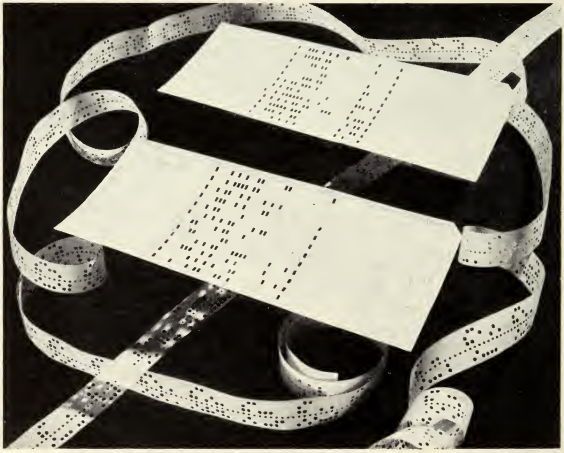
PLATE III



An English Electric DEUCE computer, built in 1955



(*a*) The magnetic drum of a DEUCE computer



(*b*) Punched cards and punched paper tape

tape a great deal faster than is usual in normal telegraphy work. Mechanical readers are in general too slow, and several photo-electric readers have been developed especially for attachment to computers. These are capable of reading speeds of up to about 200 rows of holes per second, which represents a digit rate similar to punched card equipment.

Machines using paper tape for input are normally provided with a tape perforator for output purposes. Standard telegraph punches operate at up to 15 rows of holes per second, while special high-speed perforators have been produced with speeds of up to 60 rows per second. Many machines also have a direct-coupled typewriter for printing results in a form which is immediately intelligible, and several high-speed typewriters have been developed for this purpose. The I.B.M. 701 computer, for instance, is equipped with a printer which can produce 150 lines of printing per minute, each line comprising up to 72 decimal digits.

As in the case of punched card machines, a computer installation based on paper tape will normally be provided with auxiliary equipment for preparing and copying programmes, and for tabulating numerical data.

2.43. Magnetic tape and wire

Magnetic tape has already been mentioned under the heading of auxiliary storage, but it also provides an extremely rapid means of feeding information in and out of a computer. In fact, in machines which use magnetic tape the distinction between auxiliary storage and input-output arrangements tends to become lost. Magnetic wire has also been used successfully, notably on the SEAC; its disadvantage is that only one channel is available for recording information, while on magnetic tape five or six can be used.

One drawback of magnetic material as an input-output medium is that information is not immediately visible, and the equipment needed to convert it into ordinary printing is more complicated than is the case with cards or paper tape. It is, however, extremely suitable for large-scale clerical work, since it provides a compact semi-permanent storage for material such as accounts and insurance policies.

2.44. Manual devices

In theory a computer needs hardly any manual controls except those necessary to switch it on and off. In practice, however, most

computers tend to acquire a complicated control panel. The controls normally fitted can be classified as follows:

(1) Means for switching on the computer, the auxiliary store, and the input-output units.

(2) Keys for clearing the various storage registers (i.e. setting their contents to zero). These are similar in effect to those on a desk calculating machine.

(3) Keys which cause the computer to start or stop obeying instructions. Such keys may allow the programme to proceed 'full-speed ahead', or may stop the calculation at certain pre-assigned instructions. The latter facility is useful in testing programmes.

(4) Means for feeding in small amounts of extra data. Most computers have one or more registers which are directly controlled by the operator, and which can be used to hold permanent parameters or other items of numerical information. They can also be used to make small corrections to a programme or to interpolate extra instructions. This allows the machine operator to correct programme errors without leaving the machine in order to re-perforate a piece of tape or a card.

(5) Means for displaying the contents of registers. A cathode-ray tube is usually provided for this purpose, so that the operator can inspect storage registers while the calculation is proceeding. Thus he can see whether the programme is running according to plan.

The manual controls of a computer are used mainly in testing the machine or its programmes. If both are functioning correctly, the operator may not need to touch the machine except to supply fresh data or to remove printed or punched results.

Chapter 3

THE ORGANIZATION OF PROGRAMMES

In Chapter 1 it was shown how a simple piece of numerical work could be carried out by a computer under the control of a sequence of coded instructions. Short sequences of this nature are often merely parts of larger and more complicated programmes, and when this is the case they are usually called 'routines'. This chapter describes the way in which large programmes are usually constructed, and considers some of the devices which make programming easier.

3.1. Dividing a programme into routines

The aim of programming is always to break down a calculation into a series of numerical steps which can be represented by machine instructions. In simple problems, such as the one discussed in Chapter 1, it may be possible to proceed direct from the mathematical formulation to the coded instructions, but it is often more convenient to carry out the work in two stages. The first stage consists of writing down the general plan of the calculation in ordinary human language, without considering in detail the way in which the various operations are to be carried out by the machine. Once the overall strategy has been decided, each section of the general plan is considered in detail, and the individual operations translated into machine code. The sequences for the various sections are then combined to form the complete programme.

The process is best illustrated by an example. Consider, for instance, the design of a programme to tabulate the function $\cos \sqrt{x}$ over the range $x=0$ to $x=1$, with an interval of 0·001. The first step is to write down a statement of the various operations which the programme must perform. A suitable scheme might be:

(*a*) The register which is to store the variable x is cleared. (This starts the calculation with $x=0$.)

(*b*) $\cos \sqrt{x}$ is evaluated for the stored value of x.

(*c*) x and $\cos \sqrt{x}$ are printed.

(*d*) x is increased by 0·001.

(*e*) x is tested. The programme returns to operation (*b*) if $x \leqslant 1$, but comes to a halt if $x > 1$.

Some of these steps correspond to single machine instructions, while others need sequences of instructions to carry them out. It is now possible to consider the five steps as separate coding problems, attacking each difficulty in turn.

If the programme is to be designed for the idealized machine described in Chapter 1, the only step which might cause the programmer some trouble is operation (*b*). The human computer would probably refer to a set of tables when calculating $\cos\sqrt{x}$, but, although it would be possible to store appropriate tables within the machine, it would be easier to use some kind of numerical approximation to calculate each entry in turn. One might, for instance, use the series expansion

$$\cos\sqrt{x} = 1 - \frac{x}{2!} + \frac{x^2}{4!} - \frac{x^3}{6!} \cdots,$$

taking sufficient terms to obtain accurate tabulation over the range of x considered. Terminating the series after the term in x^4 gives an approximation which may conveniently be written

$$\cos\sqrt{x} \approx 1 - x(\tfrac{1}{2} - x(\tfrac{1}{24} - x(\tfrac{1}{720} - x(\tfrac{1}{40320})))),$$

in which form it is very simple to programme. A sequence for evaluating this expression can easily be constructed using the instruction code described in Chapter 1.

If the computer to be used stores numbers in binary form, however, the operations of printing x and $\cos\sqrt{x}$ will also require sequences of instructions to convert the binary numbers into decimal ones. All that is really necessary is one sequence which prints a single number (say the content of the accumulator), since the programme can be arranged so that this sequence is used twice.

Having prepared the two sequences described above, the programmer can now proceed as though the calculation of $\cos\sqrt{x}$ and binary-decimal conversion were functions available in the machine's instruction code, like multiplication and addition. All he needs to do is to write down a controlling sequence which will carry out the other steps required, using suitable jump instructions to enter the sequences for calculating $\cos\sqrt{x}$ and for printing. Each of these can be made to end with a jump instruction which causes a return to the appropriate point in the controlling sequence. A suitable arrangement for the controlling sequence would be:

(*a*) Clear the register which is to store x.

(*b*) Jump to the sequence for calculating $\cos\sqrt{x}$.

Return from this sequence to→(*c*) Copy x into the accumulator.
(*d*) Jump to the sequence for printing the content of the accumulator.
Return from this sequence to→(*e*) Copy cos $\sqrt{x}$ into the accumulator.
(*f*) Jump to the sequence for printing the content of the accumulator.
Return from this sequence to→(*g*) Add 0·001 to the value of x.
(*h*) Form and test $(x-1)$. Return to (*b*) if $(x-1) \leqslant 0$.
(*i*) Stop.

In this arrangement of the programme, the controlling sequence described above is called a 'master routine', while the sequences for calculating cos $\sqrt{x}$ and for printing are known as 'sub-routines'. It is a characteristic feature of a sub-routine that when it has completed its task the master routine carries on from the point at which it was interrupted.

The concept of a programme as an arrangement of routines is an extremely useful one in programming. It gives the programme a logical structure, and helps the programmer to isolate and overcome singly any difficulties in coding which he may encounter. Even more important, it allows many programmes to be constructed almost entirely from pre-fabricated sub-routines. It is apparent that the short master routine described above could be used to tabulate another function merely by changing the function-generating routine, while in the same way the two sub-routines could be used in any other programme which required them. If a sub-routine is available for a particular operation, it is effectively as though the computer had been provided with an instruction for doing the job, and the programmer need not concern himself with the way in which the operation is carried out.

3.2. Library routines

When a new computer is built, the first task of its users should be to build up a library of routines for carrying out standard mathematical operations. Once constructed, these routines are always available to any programmer who may wish to use them. The efficiency of a computer depends to a large extent on the quality and comprehensiveness of its library, so that it is worth spending considerable time and effort on constructing and testing a well-integrated

set of routines. The majority of library routines fall into one of the following categories:

A. *Routines concerned with the input of information*

These are of two kinds. The first is designed to organize the transfer of instructions from the input medium to their correct places in the machine's store. If parts of the programme are to be filed away in an auxiliary store, it is this routine's job to carry out the transfers, and to start the calculation when all the programme has been stored.

The second type of input routine is used for reading decimal numbers, and in a binary machine its central feature is a decimal-binary conversion sequence. It often appears as a sub-routine called in at the beginning of a programme, and may be designed either to read a single number and place it in a certain register, or to read a series of numbers and place them in registers specified by the rest of the programme. A computer will normally have input routines for dealing with both integers and fractions.

B. *Output routines*

Output routines are mainly concerned with numerical information, since it is not often that one requires a printed record of machine instructions. Here the conversion required is from binary to decimal, and most computers are equipped with several routines. These will normally cover the output of single numbers and groups of numbers, as integers, fractions or with pre-assigned decimal point. If a machine has a direct-coupled output printer, its routines will usually include provision for tabulation in columns and pages, so that tables can be prepared with the minimum of human editing.

C. *Function routines*

These are sub-routines which generate standard mathematical functions, such as $\sqrt{x}$, $\sin x$, $\sin^{-1}x$, $\log x$, e^x, etc. The value of x is provided by the sequence which calls in the sub-routine, and it is convenient to standardize these routines so that they all use the same storage registers for x and $F(x)$. One commonly used arrangement is for the sub-routines to be entered with x available in the accumulator, and to be quitted with $F(x)$ in the same place. In the notation of §1.3 the effect of such a routine can be written simply as

$$A' = F(A).$$

Functions are normally calculated from a polynomial approximation or by an iterative process.

D. *Operator routines*

These carry out mathematical operations on functions, and normally require a sub-routine for generating the function. A typical example is a routine for the numerical integration of an arbitrary function $f(x)$. The routine is supplied with the limits of integration, and first works out suitable values x_i at which to calculate the integrand. These are then given to a sub-routine, which works out the various values $f(x_i)$. Finally, the integration routine works out an approximation to the integral from these function values, using one of the well-known formulae for numerical integration.

Another example is a routine for integrating sets of ordinary differential equations of the type

$$\frac{dx_i}{dt}=f_i(x_j\,;\,t) \quad (i,j=1\ \ldots\ n),$$

where a sub-routine is used to generate the functions appearing on the right-hand sides of the equations. An integrating routine of this type normally advances the integration by one step each time it is entered, and requires a master routine to control it.

E. *Routines for linear algebra*

These routines carry out the processes required in matrix work. The types of operation required are:

(*a*) The solution of sets of linear simultaneous algebraic equations.

(*b*) The multiplication of matrices.

(*c*) The inversion of matrices.

(*d*) The extraction of the latent roots and vectors of matrices.

With these routines, operations on matrices can be carried out almost as easily, from the programmer's point of view, as operations on single numbers. All the programmer has to do is to organize the input or calculation of the coefficients, and specify the appropriate routines.

F. *Test routines*

These routines are used to test the correct functioning of a computer. They are run before a machine is handed over by the maintenance engineers, and are available to an operator if he suspects that a machine fault has developed. A good set of test routines includes checks on each separate unit of a computer, and in some cases it is possible to locate a faulty valve merely by watching the behaviour

of the routines. The design of test routines requires considerably more knowledge of the circuitry of a machine than the average programmer possesses.

3.3. The organization of sub-routines

As mentioned in §3.1, a sub-routine always ends with a jump instruction, which causes a return to the instructions which originally called for the sub-routine. A possible arrangement is shown diagrammatically in fig. 2. It is apparent that while this allows the main body of a sub-routine to be made completely self-contained, the final

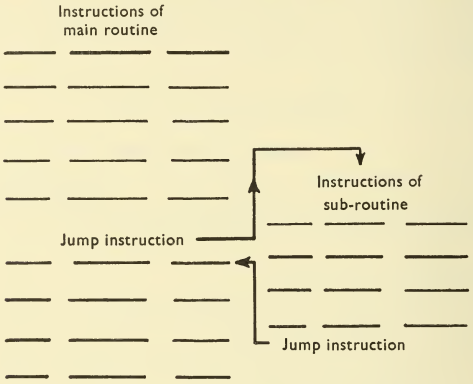


Fig. 2. Simple method of calling in a sub-routine.

jump instruction will depend on the point to which a return is required. This means that if a sub-routine is required to operate at several points in a programme, the final jump instruction must be different in each case. In the simple programme described in §3.1, for instance, the printing sub-routine is followed by a return to operation (*e*) on the first occasion, and to operation (*g*) on the second occasion. One method of achieving this is as follows.

When it is necessary to call in a sub-routine, the jump instruction in the main routine is preceded by an instruction which plants what is known as a 'link' in a convenient register. The link is merely the address of the instruction to which a return is required, and the sub-routine is arranged to terminate with the instruction 'Jump to the instruction whose address is contained in the link register'. This

arrangement is shown in fig. 3. In this way a sub-routine can be made completely independent of the instructions which call it in. It may happen that a sub-routine has also sub-routines of its own. In such a case it is necessary to keep a list of links in order to return to the main routine, but the principle is the same.

If a computer is equipped with a magnetic drum, it is often convenient to store most of the programme there. Individual routines can be stored on separate tracks, and it is possible to keep some of the more commonly used library routines permanently stored on

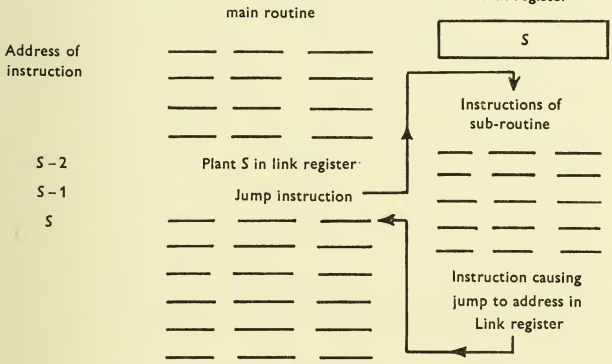


Fig. 3. Alternative method of calling in a sub-routine.

special isolated tracks whose contents cannot be over-written. In this way only a small part of the high-speed store need be reserved for instructions. Each section of the programme can be transferred from the magnetic drum as required, leaving the remainder of the high-speed store free for numerical material.

With an auxiliary store of this type, a sequence calling in a sub-routine must include appropriate instructions for transferring it to the high-speed store. Sub-routines can, however, be designed always to occupy certain registers when being obeyed. When no auxiliary store exists, all the routines have to be packed into the same store, so that when a library routine is designed it is not known in advance exactly which registers it will occupy. This difficulty is overcome on some computers by using a system of 'floating addresses', in which those

instructions whose address sections depend on the position of a routine in the store are automatically adjusted by a special input sequence as the routine is fed into the machine.

3.4. The development of a programme

It is now possible to describe what happens during the design and development of a complete programme.

The programmer's first task is to decide on a mathematical formulation for his problem. In some cases this is already fixed, but in others original mathematical research may be required. Even at this stage, however, the programmer will try, if he can, to choose the approach which leads to the simplest programming.

The second stage involves a choice of numerical method, and here the programmer is very strongly influenced by the library routines which he has available. In the example discussed in §3·1, for instance, there would almost certainly be library routines available for calculating $\sqrt{x}$ and $\cos x$. It would be simpler to make use of these than to produce a new routine for calculating $\cos \sqrt{x}$, even though the latter sequence might be slightly faster. The programmer is very often faced with a choice between designing new routines and using library routines which are not quite so well-suited to his programme. It must be remembered that even if a new routine is faster than one taken from the library, extra machine time will be necessary to test it. In the tabulation programme under discussion almost all the machine time would probably be spent on printing, so that there would be little virtue in developing a new routine. Once the mathematical problem has been reduced to a scheme of numerical analysis, the calculation can conveniently be laid out in the form of a flow-diagram, as shown in fig. 4. This helps to demonstrate the logical design of the programme and shows how the work can best be split up into routines of convenient size. At this stage the programmer decides provisionally which library routines he will use.

A start can now be made on the task of machine organization. The programmer allocates storage registers for instructions, numerical data and working space, and decides how data shall be fed in and results printed. He considers the approximate magnitudes of the numbers which will appear in the calculation, and decides whether any scaling is necessary to prevent registers overflowing or losing too many significant figures. With these details fixed, he can prepare detailed specifications of any new routines which may be required.

All that now remains is the detailed coding of the new routines,

and the final assembly of the programme in card or tape form. As each routine is prepared it is punched out and tested in the computer —usually by feeding in a few test numbers and checking the results by hand. When all the logical errors and coding slips have been eliminated, the final programme is made up by combining the separate parts on one pack of cards or a single length of tape. Master copies of library routines are normally kept available in punched form, and these can easily be copied on auxiliary editing equipment. Final testing is usually simple if all the individual routines have been checked separately.

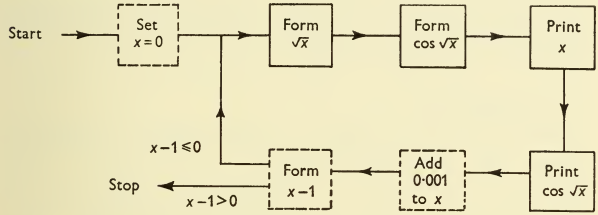


Fig. 4. Flow diagram for tabulation of cos $\sqrt{x}$. Library sub-routines shown with full outlines.

3.5. Testing programmes

Very few programmes work correctly the first time they are tried out. The most common faults in programming may be classified as follows:

A. *Slips*

These are fundamentally due to human carelessness, and careful checking of programmes will eliminate most of them. A common example is the case where a programmer writes down some instructions and then decides to change the location of a certain number. He may easily forget to alter the address parts of the relevant instructions, with the result that the computer carries out the right operations on the wrong numbers. In the same way, the rearrangement of a routine may omit an adjustment to a jump instruction, so that the instructions are obeyed in the wrong order.

Slips can also occur in transferring instructions from paper to punched form, and are often difficult to find since they do not appear on the programmer's manuscript. Here again, careful checking before the programme is fed into the machine is the only way to avoid wastage of valuable machine time.

B. *Faults of organization*

These are logical errors on the programmer's part, and usually mean that he has not foreseen all the consequences of an instruction. They may result in the programme getting stuck in a closed cycle of instructions, or in sequences of digits which were intended to represent numbers being obeyed as instructions. One fault which seems to be fairly common among users of some machines is the use of a conditional jump instruction 'the wrong way round', i.e. making a jump occur only when a number is positive, say, when it should have occurred only when the number was negative. Experience in programming reduces the number of these faults, but very few programmers avoid them entirely.

C. *Numerical faults*

These are usually the result of incorrect scaling. They result in numbers either exceeding the capacities of their registers or losing significant digits. The programme generally goes through the instructions in the correct order, but produces the wrong answer. Such faults can usually be located by making the programme print out intermediate answers at several points.

D. *Special cases*

It is usually impracticable to test a routine in all possible circumstances, and there are occasionally cases even in tested library routines where the appearance of one particular number leads to a wrong result. This may be due either to a logical fault in the routine's design or to mathematical reasons—division by zero is an obvious example. This is a particularly difficult type of fault to trace, since the programme may work correctly on one problem and fail on another.

Most computers have facilities for the inspection of their storage registers, and the simplest way of testing a programme is to make the machine go through it instruction by instruction, checking visually on the effect of each operation. With this form of testing the machine obeys a single instruction each time the operator presses a switch, so that it is extremely wasteful of machine time. One way of improving the process is to insert 'stop' instructions at suitable points in the programme. The operator can then run through the tested sections, such as library routines, at full speed, changing to single-instruction operation at the start of a suspected sequence. This

technique is not too time-wasting when carried out by an experienced operator, but it should never be considered as a substitute for the utmost care in programme preparation.

On some machines it is possible to print out a characteristic of each instruction, such as the function number, as the instruction is obeyed. In this way the programmer can quickly test whether the order of the instructions is correct. This is especially useful in checking the correctness of jump instructions. In the same way, arrangements for printing out the contents of the accumulator at specified points are extremely useful in tracing numerical faults.

It is evident that programmes constructed mainly from library routines will, on the whole, require far less time for fault tracing, and this is a strong argument for solving problems by standard mathematical processes. Machine time spent on developing a programme is just as expensive as time spent in obtaining results, and an extra complication designed to save a few seconds in a calculation may waste far more development time than it saves.

3.6. Automatic programming

The translation of instructions from normal English and mathematical symbols into machine code is a somewhat tedious and mechanical task, and it is sometimes suggested that a computer might be able to write its own programmes. In effect, this implies converting a code used by human beings into one used by a machine, and the problems which arise are similar to those occurring in problems of mechanical language translation. Programme-writing routines fall into two groups.

A. *Conversion routines*

In a conversion routine, the instructions finally placed in the machine's store are not the same as the 'instructions' punched on the input medium. These 'instructions' are written in a code which, while not as flexible as ordinary human language, is more familiar to the programmer and has wider scope than ordinary machine code. As each 'instruction' is read, the conversion routine works out the corresponding machine instruction or group of instructions, and places it in the store. When the programme of machine instructions is complete, it is obeyed by the computer in the usual way.

A very simple example of a conversion routine is the input routine mentioned in §3.3, in connexion with 'floating address' programming. Conversion routines have also been devised for doing

operations on complex numbers, and for using a computer with normal registers as though it were a floating-point machine.

B. *Interpretive routines*

With an interpretive routine, the 'instructions', which as before are written in a special code, are actually planted in the computer's store. Instead of being selected by the control unit in the normal way, however, a special sequence selects each 'instruction' in turn, decodes it, and calls in what is effectively a sub-routine for carrying out the operation specified.

Experience has so far shown that machine-produced programmes take longer to run and use more storage space than the equivalent human product. Their main advantage lies in the reduction in programming time, and the fact that virtually no testing is required. The technique is very suitable, therefore, for small problems in which the total machine time would be only a few minutes in any case. It does not seem likely, however, that machines will ever be able to carry out the broader aspects of programme design.

Chapter 4

THE SOLUTION OF ENGINEERING PROBLEMS

This chapter discusses some of the points which arise in applying computers to engineering calculations, and describes some of the problems which have so far been solved.

Although the natural way of classifying calculations is according to the basic mathematical processes involved, programme design is also considerably influenced by the number of times a calculation is likely to be repeated. In practice problems can be divided into two distinct types, which may be called 'single' and 'repetitive', and the differences between these will be considered first.

4.1. Single problems

In a single problem, the production of numerical results implies that the problem has been completely solved, and that no similar calculations will ever be required. If a programme is prepared for such a calculation, it will become redundant when the solution has been obtained, and must therefore pay for its creation during a relatively short working life.

Many research problems are of this type. The end-product of the calculation is usually a table of numbers, which may represent the solution of a differential equation, the values of a mathematical function, or standard solutions covering all practical parameter values of a particular engineering product. A certain amount of repetition may be involved, but the essential feature of a single problem is that the number of repetitions required is known beforehand.

A direct comparison of time or cost will decide whether it is better to programme such a calculation or to do it by hand on a desk machine. In assessing the effort required to solve a problem on a computer, time spent on programming and programme testing must be considered as well as actual machine running time. In many problems the task of programming takes by far the greatest part of the time, and may therefore be the determining economic factor. As mentioned in Chapter 3, both programming and programme testing are very much simplified if a calculation is reduced to standard mathematical processes for which library routines exist. Even if this

represents an 'inefficient' way of solving a problem, the use of library routines often costs less in the long run than the development of special routines which must be discarded when the calculation is complete.

A programme for a 'single' problem is like a temporary piece of laboratory apparatus, roughly built but adequate for its job. Since it will probably be used only by its creator and discarded when the job is finished, there is little point in spending a lot of time on unnecessary refinements of design.

4.2. Repetitive problems

Most engineering calculations are not, however, the complicated single problems of research. Much of the work carried out in design offices involves relatively short stereotyped calculations which occur at intervals, the same operations being carried out each time on different sets of numbers. This type of work is particularly common in firms whose products, whether steam turbines or shell roofs, are designed individually to meet particular specifications.

The development of a new design often involves analysing a series of trial designs, modifying each one in the light of previous results, and the ease with which such analyses can be carried out may determine the extent to which the final product approaches the best possible design. One problem of this type is the design of steel frameworks for buildings and bridges. Here the aim is to find the lightest and cheapest structure which will support the applied loads safely, and as long as steel structures continue to be built designers will have to carry out stress-analysis calculations, continually repeating numerical operations which are essentially the same.

A computer is extremely suitable for this type of calculation. Once a general or 'standard' programme has been constructed for a particular class of problem, each individual calculation merely involves supplying the computer with the appropriate data. No further programming is required, and the machine can in fact be operated by semi-skilled labour.

4.3. Factors in the design of standard programmes

Just as in the case of a 'single' problem, the designer of a standard programme tries to achieve his objective with the least possible effort. There are, however, several additional factors which he must consider.

One important difference springs from the fact that a standard programme, like a library routine, is intended to be of permanent

value. Any time spent in making it more efficient is therefore a capital investment which will pay dividends each time the programme is used. Even though many standard programmes are designed for relatively simple problems and take up only a few minutes of machine time, a few seconds saved may represent a considerable gain over a long period. If a programme is really going to be of general service, it is worth spending a considerable amount of time on making it as perfect as possible.

Unlike a programme for a single problem, a standard programme is always designed to cover a whole class of problems, and this means that the programmer looks for mathematical techniques which are as general as possible. With a digital computer it is very often easier to carry out an operation which in some cases may be unnecessary, than to test whether it is possible to use a short cut, while many of the intuitive judgements ordinarily made by the human computer are almost impossible to programme. Thus it often happens that the 'best' way of solving a problem on an automatic machine is one which would be quite impracticable on a desk computer, so that the programmer has to adopt new ideas as to which methods are 'easy' and which are 'difficult'.

A standard programme is essentially a production tool, which may be used by many different operators, some of whom may not be skilled programmers. It must therefore be made foolproof as regards operation and data preparation, and all 'special cases' must be foreseen. It is also most important that the programme should be designed to minimize human error. Such errors are all too common, and are often more difficult to detect than in hand-computing work. They may occur during either:

(*a*) the preparation of the initial data;

(*b*) the operation of the programme;

(*c*) the interpretation of the results.

Errors during data preparation may be minimized by arranging the programme so that the machine takes in data in the form which is most natural to the user. Small preliminary calculations, involving scaling, evaluation of coefficients, etc., are sufficiently simple to invite careless treatment, with the result that the machine may get supplied with the wrong data. For this reason it is foolish to programme the difficult parts of a calculation and neglect the easier ones. For similar reasons the programme should print the final answers in the most convenient form for the user, and not leave the last few steps to the human computer.

There is another reason why it is good policy to feed the basic parameters of a problem directly into a computer. The first part of many calculations consists of evaluating various coefficients from the initial information. These coefficients are often far more numerous than the original data, so that if they are calculated by hand outside the machine many more quantities will eventually have to be prepared for input. Whether input is by punched cards or paper tape, the smaller the volume of material the less will be the chance of mistakes in punching.

The attitude which a programmer adopts towards machine results may vary between blind faith and complete scepticism, according to the standard of reliability of his computer. Many routine engineering calculations are so short in terms of machine time that it is usually simplest to omit checking sequences, repeating the whole calculation if errors are suspected. On longer programmes of half-an-hour's duration or more it is a good idea to incorporate checks at various points, but most computers are sufficiently reliable to render complex checking sequences unjustified.

It is hardly necessary to say that a standard programme should never invite human error by requiring extensive manipulation of the machine's controls. Experience has shown that it is perfectly possible for a semi-skilled machine operator to produce satisfactory results, provided that the computer is reliable. An operator having no knowledge of the construction of a programme is naturally at a disadvantage if suspect results appear, as he may be unable to tell whether the fault lies in the machine, the programme, or the punching of the data.

The first standard programmes were often merely direct translations into machine code of operations previously carried out by hand. They were therefore 'small' calculations from the computer point of view, and required only a few minutes of machine time. It seems likely, however, that as engineers realize the power of automatic computers the size of standard programmes will increase, until calculations which would be quite impracticable to carry out by hand become 'standard practice'.

A standard programme can be compared with a machine tool designed for mass-production work. Just as a machine tool is redesigned in the light of experience, so a standard programme is often gradually modified as experience of using it accumulates.

4.4. Problems which have been solved on computers

This section does not attempt to describe all the problems which have so far been tackled. It merely gives examples of the kind of work which is being done at present on digital machines.

4.41. Problems in linear algebra

Many engineering calculations are concerned with linear systems, built up from elements which obey Hooke's or Ohm's law, and in such systems there are usually two kinds of variable which are related in a linear manner. In an elastic structure, for instance, the displacements are linear functions of the applied loads, while in an electric network currents are similarly related to applied voltages.

If such a calculation involves finding the static response to a given applied stimulus, it can always be formulated as a set of simultaneous linear equations, while if the general static response to any arbitrary applied stimulus is required, the solution involves the inversion of a matrix. In the same way, the common problem of finding natural frequencies becomes from the mathematical point of view the extraction of the latent roots of a matrix. Solving such a problem on a computer merely involves calling in the appropriate library routine. In recent years a great deal of effort has been spent on developing techniques for dealing with large sets of equations, so that the programmer can consider his problem solved if he can reduce it to one of the three standard matrix processes mentioned above. The Manchester computer, for example, which is not a particularly fast machine, has a library routine which solves n simultaneous equations ($n < 64$) in approximately $n^2/6$ sec., the inversion of the associated matrix taking about three times as long. Other machines have routines of comparable speed and capacity. Larger sets of equations do not often occur, but when they do it is usually possible to break the problem down into several smaller problems of more manageable size.

The fact that a linear problem can be reduced to a standard mathematical form does not mean that a computer is only of use in solving the equations. If the problem is a 'single' one it may be best to work out the coefficients by hand, but in a repetitive problem it is most desirable to use the machine to set up the equations as well as to solve them. A set of 50 simultaneous equations, for instance, may have up to 2500 non-zero coefficients, and if all these are calculated by hand it will be a major task merely to check the punching. The remarks on the avoidance of human error in §4.3 are very relevant here.

The programmer's main problem in dealing with repetitive linear problems is therefore the design of routines for translating the data of a physical system into a set of equations. One example of such routines occurs in a standard programme for elastic structural analysis developed on the Manchester computer. In this programme, the machine is first supplied with the dimensions and section constants of the various members of a framework, from which it calculates the stiffnesses of the individual members. The data tape then specifies the applied loads and the way in which the members are connected together, and the programme uses this information, together with the stiffnesses previously calculated, to set up the load-displacement equations for the complete structure. After a library routine has solved these equations for the displacements, the internal forces and moments in the members are calculated and printed out in order. The programme can be applied to any rigid-jointed plane framework, the complete analysis of a frame with 10 joints taking about 8 min. A similar technique has been applied to problems of electric network analysis.

Another problem which was solved on the Manchester computer by reducing it to a set of simultaneous linear equations was the determination of the elastic stress distribution in a transversely loaded square floor supported by edge-beams, for various symmetric and unsymmetric conditions of loading and support. Mathematically the problem is one of solving the bi-harmonic equation in a square region, with appropriate boundary conditions at the edges, where the floor-slab is attached to the supporting joists. The method adopted was to replace the deflexion of the slab by the deflexions at the nodes of a 7×7 square mesh, using finite difference techniques to set up the equations relating the displacements of the mesh points to the applied loads. Here again the problem was the setting up of the 49 simultaneous equations; the final programme did this automatically, as well as calculating the values of the bending and twisting moments at the mesh points. Computing times for the various parts of the calculation were:

Reading the values of beam/slab stiffness for the four edge-beams, the loads at the mesh points, and setting up the 49 difference equations	50 sec.
Solving the equations and printing the displacements at the mesh points	5 min.
Calculating the bending and twisting moments	4 min. (mostly printing)

On one occasion 25 different cases were analysed in $4\frac{3}{4}$ hr. It is interesting to note that the formation of the equations inside the machine took considerably less time than would have been required for reading the equations from tape.

Another common problem for which several standard programmes exist is the determination of natural frequencies of torsional and transverse vibration in turbine shafts and other power transmission systems. Although this can be reduced to a latent-root problem, the special characteristics of such systems allow other methods to be used. These are based on determining the exciting force needed to maintain vibration at a number of frequencies, the natural frequencies corresponding to the zeros of the forcing term. With a computer it is a very simple matter to carry out such a calculation for a large number of frequencies, thus determining the response of a system over the whole working range.

4.42. Linear programming

Problems of analysis can often be reduced to the solution of sets of linear equations, but problems of design give rise to a different type of calculation.

The technical problem of satisfying a certain set of design requirements can usually be met in an infinite number of different ways, and it is the designer's job to find the solution which, in some sense or other, is the 'best' one. If the design problem is a linear one, it can sometimes be reduced to the mathematical problem of determining the values of a set of variables which minimize a known linear function, while satisfying a set of linear inequalities. The linear function may represent the cost of an article, for instance, while the inequalities represent certain specifications which it must satisfy.

In designing a steel structure to support a given loading system, for example, the cost of the structure may be considered to be roughly proportional to its weight. If the design is based on the criterion of plastic collapse, the fact that the structure must not fail in any one of all the possible collapse mechanisms defines a set of linear inequalities which must be satisfied by the fully plastic moments of the members. The designer's task is to assign dimensions to the various members in such a way that these conditions are satisfied, while at the same time minimizing the total weight and cost. Such a problem is called a 'linear programming' problem—the word 'programming' being used here not in the sense of a computer programme, but rather with the meaning 'planning'.

In all but the most trivial cases the exact solution of a linear programming problem is only practicable on a computer, and most of the numerical techniques which have so far been developed have been designed for machine use. Although in many cases common-sense may give a solution which is very near to the correct one, a saving of a small percentage may yet represent quite a considerable amount in a large-scale design, so that digital computers may have a useful part to play in this type of work. Linear programming techniques may also be used to solve problems arising in the economic planning of other human activities such as transport, marketing, etc.

4.43. Ordinary differential equations

Computers have been applied successfully to many problems involving the numerical integration of differential equations. By introducing extra variables, where necessary, any set of ordinary simultaneous differential equations can be reduced to a set of n first-order equations of the form

$$\frac{dx_i}{dt}=f_i(x_j;\, t) \quad (i, j=1, 2, \ldots, n).$$

If the values of the variables x_i are all known for one value of t, the equations can be integrated forward (or backward) in t by any standard numerical step-by-step process. The Runge-Kutta method is perhaps the one most commonly met with in computer programmes, as it does not require a series of starting values and allows the step-length to be changed without difficulty, but other methods have also been used successfully.

As mentioned previously, the programmer normally has a library routine available for carrying out the integration. In most computer libraries this is designed to advance the integration by one step each time it is called in, and a master routine is required for controlling the length and number of the steps. The programmer also has to provide sub-routines for calculating the appropriate f_i's at each stage.

In many cases the equations to be integrated are equations of motion. Problems of this type which have been solved on computers include trajectory calculations for electrons, projectiles and the satellites of binary stars. In each case the use of a computer allows the calculation of any number of trajectories, the operator merely having to provide the machine with the appropriate series of starting conditions. A slightly different type of problem recently solved at Manchester involved the calculation of surge voltages in a transformer

winding. By treating the physical system as an inductance-capacitance network, the problem was reduced to the forward integration of a set of forty ordinary differential equations of the type described above.

In cases where the boundary conditions are given at two ends of an interval a process of trial and error is normally required. Starting values which are not known are assumed, and the integration carried forward to the other end of the interval. The remaining boundary conditions are then compared with the values calculated, and an adjustment made to the assumed initial conditions. The process is continued until satisfactory agreement is reached at both ends of the interval. If the functions f_i are linear in the x's, an alternative approach is to replace all differential coefficients by differences. This has the effect of reducing the problem to the solution of a set of simultaneous linear algebraic equations.

4.44. Partial differential equations

Partial differential equations with two independent variables may be treated in a number of ways, according to the equations and the boundary conditions. Problems requiring a solution of Laplace's, Poisson's or the bi-harmonic equation, in which boundary values are specified on the edge of a closed region, are often treated by reducing all differential coefficients to differences, and solving the resulting algebraic equations by relaxation or by direct means. An example of this approach is the floor-slab problem mentioned in §4.41. For hyperbolic equations such as the wave equation, on the other hand, it is more usual to remove only one independent variable by using finite differences, integrating the resulting set of ordinary differential equations in the manner described above.

Equations with more than two independent variables, such as arise in unsteady two- or three-dimensional heat or fluid flow problems, can be treated by similar means. In general, they require a very fast machine with a large amount of storage.

4.45. Problems in non-linear algebra

Non-linear problems are in general more difficult than linear ones, and the possibilities of digital computers are correspondingly greater.

Many non-linear problems can be solved by iterative means, each iteration being a linear calculation. An example of this technique is an extension of the structural analysis programme described in §4.41 to cover the non-linear stability effects of axial forces in

members. A preliminary linear elastic analysis calculates approximate values for the axial forces, and these are then used to modify the bending stiffnesses of the members, in accordance with well-known theory. The calculation is then repeated with these modified stiffnesses, the cycle being repeated until a sufficiently accurate answer has been obtained.

Another interesting non-linear problem is the design of optical lenses. Ray-tracing through an optical system is merely a matter of coordinate geometry, normal refraction laws being used to find the change of direction at each surface, and with an automatic computer it is possible to trace the path of a single ray through a complete lens system in a fraction of a second. In one design programme at present in use, a series of rays are traced, and the information gained is used to evaluate a 'performance function', which measures the amount of distortion, coma, etc., of the complete system. The routines for this part of the calculation appear as sub-routines in a larger programme, which systematically tries to minimize the performance function by altering the design constants of the various lenses. With this programme a two-hour run achieved considerable improvement in a design of the 'Tessar' type which has been in use for many years. Work is at present in progress on similar programmes for the design of magnetic lenses for focusing electron beams.

4.46. Function tabulation

Computers rarely use tables of functions themselves, since they can so easily be programmed to generate most of the functions they may require, but they are nevertheless extremely suitable for preparing tables. Perhaps the most famous set produced on a machine are the Harvard tables of Bessel functions, but many other useful tables have been produced elsewhere. By incorporating column and page lay-out in the output routine of a programme it is possible to produce tables with the minimum of human intervention and error.

4.47. Non-mathematical problems

One important application of digital computers is to commercial data-processing and accountancy calculations. This type of work requires a great deal of input and output with very little intervening arithmetic, so that the type of machine designed for scientific calculations is not particularly suitable. Specially designed machines are, however, playing an increasingly important part in mechanizing office procedure.

On the more frivolous side, machines have been programmed to play chess and draughts, and to make opening bids at bridge. They have not, however, so far evinced any marked superiority over human beings at these games.

4.5. Digital computers and mathematical methods

It will be realized that the problems facing the programmer are quite different from those facing the user of a desk machine. The human computer is always concerned with minimizing the total amount of arithmetic he has to do, and to do this he draws extensively on intuition and experience. The programmer, on the other hand, tries to reduce his calculation to a simple logical process, and makes as much use as he can of standard mathematical techniques.

The difference in the two approaches is particularly marked in engineering problems, where mathematical analysis is often developed solely as a basis for subsequent numerical work. The more often a problem has been solved on a desk machine, the more certain it is that the methods developed need re-examining before translation into a computer programme. An example of this may be found in the mathematical technique of 'relaxation' and the comparable methods of 'moment-distribution' in structural analysis. Both these techniques were developed because classical methods of formulating linear problems led to large sets of simultaneous equations, the direct solution of which was tedious by hand. Indeed, one might say that most of the methods developed for elastic structural analysis are really devices for avoiding the formal solution of large sets of linear equations. It is rather strange to find that with an automatic computer available one often actually tries to reduce a problem to this form.

When a problem is prepared for hand computing, the mathematical treatment often includes analytical transformations which, while they may be formally elegant, result in the calculation being carried out with numbers which have no direct physical significance. The programmer, on the other hand, tends to keep close to the basic equations of a physical system, even if this leads to a slightly longer calculation. Mathematical analysis developed as a prelude to a computer programme may thus help to throw fresh light on the nature of a problem, while the possibility of using a machine to obtain rapid exact solutions has already stimulated fresh theoretical work on problems previously treated empirically.

4.6. The human factor

The engineer who wishes to use a digital computer, but who has not sufficient numerical work to justify the purchase of a machine, has two courses open to him. He can either learn to programme himself, hiring time as required on one of the machines now available, or he can employ a commercial computing organization to solve his problems for him. Learning to programme may take him a couple of months, but if the work which he requires can be done by an existing standard programme a few days' training in data preparation and machine operation is all that is required. Hire charges for large machines range from £20 to £50 per hour. This may appear high, but in fact it represents a considerable economy in many problems. Once a programme has been completed, the speed of a computer is from 100 to 500 times that of a human being equipped with a desk calculating machine.

From the remarks made in §4.5 it is apparent that the programmer may require on occasion considerably more than the mere ability to translate mathematical processes into machine code. To take full advantage of his machine he needs sufficient vision and mathematical skill to design new methods of solving old problems, while if he is dealing with engineering calculations he also needs a sound background of engineering knowledge. At present there is a serious shortage of engineers who have had a mathematical training suitable for programming work.

It is often assumed that an automatic computer is only useful in solving large problems. This may be true in the case of 'single' calculations, but it is doubtful if the assumption holds for problems which can be treated by standard programmes. In many cases a decision as to which is the 'best' method of attack depends not so much on the relative merits of hand and automatic machines as on the relative abilities of the human beings available to use them. The consequences of human incompetence are amplified rather than concealed by the speed of an electronic machine, and it is vitally important that the human organization built round a computer should be able to make the most of its potentialities. A machine may be able to do in an hour what would take a human being six months, but the value of what is done in that hour remains entirely a human responsibility.

4.7. Future prospects

The early machines were relatively slow, used large quantities of equipment, and were often unreliable. Fortunately, however, computer development followed a pattern familiar in other branches of engineering. Speed and reliability improved, while cost, size and power consumption decreased. The open racks and trailing wires of the laboratory-built machines gradually gave way to the enamelled cabinets and neatly arranged control panels of their factory-built successors. Many different types of computer are now on the market, ranging from very fast machines with large storage capacities designed for general scientific calculations, to simple computers with rudimentary programming facilities intended for use in accountancy and other clerical work.

While the engineering side of computers has progressed, mathematicians have been exploring the potentialities of the new machines, and developing the art of programming. What was at first an abstruse specialism only practised by a few people has become a subject in which regular courses are held at many universities. Programming itself has become easier; this is partly because of accumulated experience, and partly because modern machines offer many facilities to the programmer which their predecessors lacked. On some computers learning to programme now takes only a few weeks, as compared with the three or four months necessary on earlier machines.

The computers of the future will doubtless be better than those available at present, but the changes will probably be ones of detailed design rather than of fundamental principle. Magnetic cores are already coming to be regarded as the standard elements for the high-speed stores of large machines, while transistors are also likely to play an important part in future designs.

Developments are more likely to be spectacular in the field of applications. On the research side there are many problems, particularly in the field of partial differential equations, which tax severely the speed and capacity of the largest machines yet built. At a lower level, the development of standard programmes similar to those described in this chapter may lead to radical changes in the routine calculations carried out in industry. Another new and interesting application of computers is in the control of industrial and commercial processes. The idea of an automatic factory directly controlled by a special-purpose computer no longer belongs exclusively to the realms of science fiction, while the mechanization of clerical

work in large offices has created a demand for medium-sized computers, usually with complex input-output equipment.

The digital computer is now firmly established among the many other tools available to the engineer, and it is natural that this fact should be reflected in his education. This does not mean that every student should be taught how to programme—programming is after all merely a technique which can be learnt fairly quickly when required. It is far more important that he should know something of the mathematical techniques most useful in computer work, so that later on he will be in a position to judge whether his own particular problems are suitable for machine solution.

REFERENCES FOR FURTHER READING

The following are merely a selection from the literature available:

General principles

Bowden, B. V. (Ed.). *Faster than Thought* (Pitman, 1953). A popular account of digital computers, including a general survey of their present and future applications.

Hartree, D. R. *Calculating Instruments and Machines* (Cambridge University Press, 1950). A general introduction also covering certain types of analogue computer.

Booth, A. D. and Booth, K. H. V. *Automatic Digital Calculators* (Butterworth, 1953).

Wilkes, M. V. *Automatic Digital Computers* (Methuen, 1956).

These two books are more technical and are largely concerned with details of machine design. A few applications are described.

Applications to particular problems

Brooker, R. A. Application of digital computing techniques to physics. *Brit. J. Appl. Phys.* vol. 4, p. 321 (November 1953).

Hunt, P. M. The digital computer in aircraft structural analysis. *Aeronaut. Engng*, vol. 28, p. 70 (March 1956).

Livesley, R. K. The application of a digital computer to some problems of structural analysis. *Struct. Engr*, vol. 34, p. 1 (January 1956).

Locke, W. N. and Booth, A. D. *The Machine Translation of Languages* (John Wiley, New York, and Chapman and Hall, London, 1955).

Convention on Digital Computer techniques, London, April 1956. *Proc. Instn Elect. Engrs*, vol. 103, part B, Supplement no. 1. The section entitled 'Engineering and Scientific Applications' contains nine papers describing the application of computers to electrical engineering problems.